第二十八章 流程控制：if分支结构

In the last chapter, we were presented with a problem. How can we make our report generator script adapt to the privileges of the user running the script? The solution to this problem will require us to find a way to “change directions” within our script, based on a the results of a test. In programming terms, we need the program to branch. Let’s consider a simple example of logic expressed in pseudocode, a simulation of a computer language intended for human consumption:

在上一章中，我们遇到一个问题。怎样使我们的报告生成器脚本能适应运行此脚本的用户的权限？ 这个问题的解决方案要求我们能找到一种方法，在脚本中基于测试条件结果，来“改变方向”。 用编程术语表达，就是我们需要程序可以分支。让我们考虑一个简单的用伪码表示的逻辑实例， 伪码是一种模拟的计算机语言，为的是便于人们理解：

X=5

If X = 5, then:

Say “X equals 5.”

Otherwise:

Say “X is not equal to 5.”

This is an example of a branch. Based on the condition, “Does X = 5?” do one thing, “Say X equals 5,” otherwise do another thing, “Say X is not equal to 5.”

这就是一个分支的例子。根据条件，“Does X = 5?” 做一件事情，“Say X equals 5,” 否则，做另一件事情，“Say X is not equal to 5.”

**if**

Using the shell, we can code the logic above as follows:

使用 shell，我们可以编码上面的逻辑，如下所示：

x=5

if [ $x = 5 ]; then

echo "x equals 5."

else

echo "x does not equal 5."

fi

or we can enter it directly at the command line (slightly shortened):

或者我们可以直接在命令行中输入以上代码（略有缩短）：

[me@linuxbox ~]$ x=5

[me@linuxbox ~]$ if [ $x = 5 ]; then echo "equals 5"; else echo "does

not equal 5"; fi

equals 5

[me@linuxbox ~]$ x=0

[me@linuxbox ~]$ if [ $x = 5 ]; then echo "equals 5"; else echo "does

not equal 5"; fi

does not equal 5

In this example, we execute the command twice. Once, with the value of x set to 5, which results in the string “equals 5” being output, and the second time with the value of x set to 0, which results in the string “does not equal 5” being output.

在这个例子中，我们执行了两次这个命令。第一次是，把 x 的值设置为5，从而导致输出字符串“equals 5”, 第二次是，把 x 的值设置为0，从而导致输出字符串“does not equal 5”。

The if statement has the following syntax:

这个 if 语句语法如下：

if commands; then

commands

[elif commands; then

commands...]

[else

commands]

fi

where commands is a list of commands. This is a little confusing at first glance. But before we can clear this up, we have to look at how the shell evaluates the success or failure of a command.

这里的 commands 是指一系列命令。第一眼看到会有点儿困惑。但是在我们弄清楚这些语句之前，我们 必须看一下 shell 是如何评判一个命令的成功与失败的。

**退出状态**

Commands (including the scripts and shell functions we write) issue a value to the system when they terminate, called an exit status. This value, which is an integer in the range of 0 to 255, indicates the success or failure of the command’s execution. By convention, a value of zero indicates success and any other value indicates failure. The shell provides a parameter that we can use to examine the exit status. Here we see it in action:

当命令执行完毕后，命令（包括我们编写的脚本和 shell 函数）会给系统发送一个值，叫做退出状态。 这个值是一个 0 到 255 之间的整数，说明命令执行成功或是失败。按照惯例，一个零值说明成功，其它所有值说明失败。 Shell 提供了一个参数，我们可以用它检查退出状态。用具体实例看一下：

[me@linuxbox ~]$ ls -d /usr/bin

/usr/bin

[me@linuxbox ~]$ echo $?

0

[me@linuxbox ~]$ ls -d /bin/usr

ls: cannot access /bin/usr: No such file or directory

[me@linuxbox ~]$ echo $?

2

In this example, we execute the ls command twice. The first time, the command executes successfully. If we display the value of the parameter $?, we see that it is zero. We execute the ls command a second time, producing an error and examine the parameter $? again. This time it contains a 2, indicating that the command encountered an error. Some commands use different exit status values to provide diagnostics for errors, while many commands simply exit with a value of one when they fail. Man pages often include a section entitled “Exit Status,” describing what codes are used. However, a zero always indicates success.

在这个例子中，我们执行了两次 ls 命令。第一次，命令执行成功。如果我们显示参数$?的值，我们 看到它是零。我们第二次执行 ls 命令的时候，产生了一个错误，并再次查看参数$?。这次它包含一个 数字 2，表明这个命令遇到了一个错误。有些命令使用不同的退出值，来诊断错误，而许多命令当 它们执行失败的时候，会简单地退出并发送一个数字1。手册页中经常会包含一章标题为“退出状态”的内容， 描述了使用的代码。然而，一个零总是表明成功。

The shell provides two extremely simple builtin commands that do nothing except terminate with either a zero or one exit status. The true command always executes successfully and the false command always executes unsuccessfully:

这个 shell 提供了两个极其简单的内部命令，它们不做任何事情，除了以一个零或1退出状态来终止执行。 True 命令总是执行成功，而 false 命令总是执行失败：

[me@linuxbox~]$ true

[me@linuxbox~]$ echo $?

0

[me@linuxbox~]$ false

[me@linuxbox~]$ echo $?

1

We can use these commands to see how the if statement works. What the if statement really does is evaluate the success or failure of commands:

我们能够使用这些命令，来看一下 if 语句是怎样工作的。If 语句真正做的事情是计算命令执行成功或失败：

[me@linuxbox ~]$ if true; then echo "It's true."; fi

It's true.

[me@linuxbox ~]$ if false; then echo "It's true."; fi

[me@linuxbox ~]$

The command echo “It’s true.” is executed when the command following if executes successfully, and is not executed when the command following if does not execute successfully. If a list of commands follows if, the last command in the list is evaluated:

当 if 之后的命令执行成功的时候，命令 echo “It’s true.” 将会执行，否则此命令不执行。 如果 if 之后跟随一系列命令，则将计算列表中的最后一个命令：

[me@linuxbox ~]$ if false; true; then echo "It's true."; fi

It's true.

[me@linuxbox ~]$ if true; false; then echo "It's true."; fi

[me@linuxbox ~]$

**测试**

By far, the command used most frequently with if is test. The test command performs a variety of checks and comparisons. It has two equivalent forms:

到目前为止，经常与 if 一块使用的命令是 test。这个 test 命令执行各种各样的检查与比较。 它有两种等价模式：

test expression

and the more popular:

比较流行的格式是：

[ expression ]

where expression is an expression that is evaluated as either true or false. The test command returns an exit status of zero when the expression is true and a status of one when the expression is false.

这里的 expression 是一个表达式，其执行结果是 true 或者是 false。当表达式为真时，这个 test 命令返回一个零 退出状态，当表达式为假时，test 命令退出状态为1。

**文件表达式**

The following expressions are used to evaluate the status of files:

以下表达式被用来计算文件状态：

|  |  |
| --- | --- |
| *Table 28-1: test File Expressions* | |
| Expression | Is Ture If |
| file1 -ef file2 | file1 and file2 have the same inode numbers (the two filenames refer to the same file by hard linking). |
| file1 -nt file2 | file 1 is newer than file2. |
| file1 -ot file2 | file1 is older than file2. |
| -b file | file exists and is a block special (device) file. |
| -c file | file exists and is a character special (device) file. |
| -d file | file exists and is a directory. |
| -e file | file exists. |
| -f file | file exists and is a regular file. |
| -g file | file exists and is set-group-ID. |
| -G file | file exists and is owned by the effective group ID. |
| -k file | file exists and has its “sticky bit” set. |
| -L file | file exists and is a symbolic link. |
| -O file | file exists and is owned by the effective user ID. |
| -p file | file exists and is a named pipe. |
| -r file | file exists and is readable (has readable permission for the effective user). |
| -s file | file exists and has a length greater than zero. |
| -S file | file exists and is a network socket. |
| -t fd | fd is a file descriptor directed to/from the terminal. This can be used to determine whether standard input/output/ error is being redirected. |
| -u file | file exists and is setuid. |
| -w file | file exists and is writable (has write permission for the effective user). |
| -x file | file exists and is executable (has execute/search permission for the effective user). |

|  |  |
| --- | --- |
| *表28-1: 测试文件表达式* | |
| 表达式 | 如果为真 |
| file1 -ef file2 | file1 和 file2 拥有相同的索引号（通过硬链接两个文件名指向相同的文件）。 |
| file1 -nt file2 | file1新于 file2。 |
| file1 -ot file2 | file1早于 file2。 |
| -b file | file 存在并且是一个块（设备）文件。 |
| -c file | file 存在并且是一个字符（设备）文件。 |
| -d file | file 存在并且是一个目录。 |
| -e file | file 存在。 |
| -f file | file 存在并且是一个普通文件。 |
| -g file | file 存在并且设置了组 ID。 |
| -G file | file 存在并且由有效组 ID 拥有。 |
| -k file | file 存在并且设置了它的“sticky bit”。 |
| -L file | file 存在并且是一个符号链接。 |
| -O file | file 存在并且由有效用户 ID 拥有。 |
| -p file | file 存在并且是一个命名管道。 |
| -r file | file 存在并且可读（有效用户有可读权限）。 |
| -s file | file 存在且其长度大于零。 |
| -S file | file 存在且是一个网络 socket。 |
| -t fd | fd 是一个定向到终端／从终端定向的文件描述符 。 这可以被用来决定是否重定向了标准输入／输出错误。 |
| -u file | file 存在并且设置了 setuid 位。 |
| -w file | file 存在并且可写（有效用户拥有可写权限）。 |
| -x file | file 存在并且可执行（有效用户有执行／搜索权限）。 |

Here we have a script that demonstrates some of the file expressions:

这里我们有一个脚本说明了一些文件表达式：

#!/bin/bash

# test-file: Evaluate the status of a file

FILE=~/.bashrc

if [ -e "$FILE" ]; then

if [ -f "$FILE" ]; then

echo "$FILE is a regular file."

fi

if [ -d "$FILE" ]; then

echo "$FILE is a directory."

fi

if [ -r "$FILE" ]; then

echo "$FILE is readable."

fi

if [ -w "$FILE" ]; then

echo "$FILE is writable."

fi

if [ -x "$FILE" ]; then

echo "$FILE is executable/searchable."

fi

else

echo "$FILE does not exist"

exit 1

fi

exit

The script evaluates the file assigned to the constant FILE and displays its results as the evaluation is performed. There are two interesting things to note about this script. First, notice how the parameter$FILE is quoted within the expressions. This is not required, but is a defense against the parameter being empty. If the parameter expansion of $FILE were to result in an empty value, it would cause an error (the operators would be interpreted as non-null strings rather than operators). Using the quotes around the parameter insures that the operator is always followed by a string, even if the string is empty. Second, notice the presence of the exit commands near the end of the script. The exit command accepts a single, optional argument, which becomes the script’s exit status. When no argument is passed, the exit status defaults to zero. Using exit in this way allows the script to indicate failure if $FILE expands to the name of a nonexistent file. The exit command appearing on the last line of the script is there as a formality. When a script “runs off the end” (reaches end of file), it terminates with an exit status of zero by default, anyway.

这个脚本会计算赋值给常量 FILE 的文件，并显示计算结果。对于此脚本有两点需要注意。第一个， 在表达式中参数$FILE是怎样被引用的。引号并不是必需的，但这是为了防范空参数。如果$FILE的参数展开 是一个空值，就会导致一个错误（操作符将会被解释为非空的字符串而不是操作符）。用引号把参数引起来就 确保了操作符之后总是跟随着一个字符串，即使字符串为空。第二个，注意脚本末尾的 exit 命令。 这个 exit 命令接受一个单独的，可选的参数，其成为脚本的退出状态。当不传递参数时，退出状态默认为零。 以这种方式使用 exit 命令，则允许此脚本提示失败如果 $FILE 展开成一个不存在的文件名。这个 exit 命令 出现在脚本中的最后一行，是一个当一个脚本“运行到最后”（到达文件末尾），不管怎样， 默认情况下它以退出状态零终止。

Similarly, shell functions can return an exit status by including an integer argument to the return command. If we were to convert the script above to a shell function to include it in a larger program, we could replace the exit commands with return statements and get the desired behavior:

类似地，通过带有一个整数参数的 return 命令，shell 函数可以返回一个退出状态。如果我们打算把 上面的脚本转变为一个 shell 函数，为了在更大的程序中包含此函数，我们用 return 语句来代替 exit 命令， 则得到期望的行为：

test\_file () {

# test-file: Evaluate the status of a file

FILE=~/.bashrc

if [ -e "$FILE" ]; then

if [ -f "$FILE" ]; then

echo "$FILE is a regular file."

fi

if [ -d "$FILE" ]; then

echo "$FILE is a directory."

fi

if [ -r "$FILE" ]; then

echo "$FILE is readable."

fi

if [ -w "$FILE" ]; then

echo "$FILE is writable."

fi

if [ -x "$FILE" ]; then

echo "$FILE is executable/searchable."

fi

else

echo "$FILE does not exist"

return 1

fi

}

**字符串表达式**

The following expressions are used to evaluate strings:

以下表达式用来计算字符串：

|  |  |
| --- | --- |
| *Table 28-2: test String Expressions* | |
| Expression | Is Ture If... |
| string | string is not null. |
| -n string | The length of string is greater than zero. |
| -z string | The length of string is zero. |
| string1 = string2  string1 == string2 | string1 and string2 are equal. Single or double equal signs may be used, but the use of double equal signs is greatly preferred. |
| string1 != string2 | string1 and string2 are not equal. |
| string1 > string2 | sting1 sorts after string2. |
| string1 < string2 | string1 sorts before string2. |

|  |  |
| --- | --- |
| *表28-2: 测试字符串表达式* | |
| 表达式 | 如果为真... |
| string | string 不为 null。 |
| -n string | 字符串 string 的长度大于零。 |
| -z string | 字符串 string 的长度为零。 |
| string1 = string2  string1 == string2 | string1 和 string2 相同. 单或双等号都可以，不过双等号更受欢迎。 |
| string1 != string2 | string1 和 string2 不相同。 |
| string1 > string2 | sting1 排列在 string2 之后。 |
| string1 < string2 | string1 排列在 string2 之前。 |

Warning: the > and < expression operators must be quoted (or escaped with a backslash) when used with test. If they are not, they will be interpreted by the shell as redirection operators, with potentially destructive results. Also note that while the bash documentation states that the sorting order conforms to the collation order of the current locale, it does not. ASCII (POSIX) order is used in versions of bash up to and including 4.0.

警告：这个 > 和 <表达式操作符必须用引号引起来（或者是用反斜杠转义）， 当与 test 一块使用的时候。如果不这样，它们会被 shell 解释为重定向操作符，造成潜在地破坏结果。 同时也要注意虽然 bash 文档声明排序遵从当前语系的排列规则，但并不这样。将来的 bash 版本，包含 4.0， 使用 ASCII（POSIX）排序规则。

Here is a script that demonstrates them:

这是一个演示这些问题的脚本：

#!/bin/bash

# test-string: evaluate the value of a string

ANSWER=maybe

if [ -z "$ANSWER" ]; then

echo "There is no answer." >&2

exit 1

fi

if [ "$ANSWER" = "yes" ]; then

echo "The answer is YES."

elif [ "$ANSWER" = "no" ]; then

echo "The answer is NO."

elif [ "$ANSWER" = "maybe" ]; then

echo "The answer is MAYBE."

else

echo "The answer is UNKNOWN."

fi

In this script, we evaluate the constant ANSWER. We first determine if the string is empty. If it is, we terminate the script and set the exit status to one. Notice the redirection that is applied to the echo command. This redirects the error message “There is no answer.” to standard error, which is the “proper” thing to do with error messages. If the string is not empty, we evaluate the value of the string to see if it is equal to either “yes,” “no,” or “maybe.” We do this by using elif, which is short for “else if.” By using elif, we are able to construct a more complex logical test.

在这个脚本中，我们计算常量 ANSWER。我们首先确定是否此字符串为空。如果为空，我们就终止 脚本，并把退出状态设为零。注意这个应用于 echo 命令的重定向操作。其把错误信息 “There is no answer.” 重定向到标准错误，这是处理错误信息的“合理”方法。如果字符串不为空，我们就计算 字符串的值，看看它是否等于“yes,” “no,” 或者“maybe”。为此使用了 elif，它是 “else if” 的简写。 通过使用 elif，我们能够构建更复杂的逻辑测试。

**整型表达式**

The following expressions are used with integers:

下面的表达式用于整数：

|  |  |
| --- | --- |
| *Table 28-3: test Integer Expressions* | |
| Expression | Is True If... |
| integer1 -eq integer2 | integer1 is equal to integer2. |
| integer1 -ne integer2 | integer1 is not equal to integer2. |
| integer1 -le integer2 | integer1 is less than or equal to integer2. |
| integer1 -lt integer2 | integer1 is less than integer2. |
| integer1 -ge integer2 | integer1 is greater than or equal to integer2. |
| integer1 -gt integer2 | integer1 is greater than integer2. |

|  |  |
| --- | --- |
| *表28-3: 测试整数表达式* | |
| 表达式 | 如果为真... |
| integer1 -eq integer2 | integer1 等于 integer2. |
| integer1 -ne integer2 | integer1 不等于 integer2. |
| integer1 -le integer2 | integer1 小于或等于 integer2. |
| integer1 -lt integer2 | integer1 小于 integer2. |
| integer1 -ge integer2 | integer1 大于或等于 integer2. |
| integer1 -gt integer2 | integer1 大于 integer2. |

Here is a script that demonstrates them:

这里是一个演示以上表达式用法的脚本：

#!/bin/bash

# test-integer: evaluate the value of an integer.

INT=-5

if [ -z "$INT" ]; then

echo "INT is empty." >&2

exit 1

fi

if [ $INT -eq 0 ]; then

echo "INT is zero."

else

if [ $INT -lt 0 ]; then

echo "INT is negative."

else

echo "INT is positive."

fi

if [ $((INT % 2)) -eq 0 ]; then

echo "INT is even."

else

echo "INT is odd."

fi

fi

The interesting part of the script is how it determines whether an integer is even or odd. By performing a modulo 2 operation on the number, which divides the number by two and returns the remainder, it can tell if the number is odd or even.

这个脚本中有趣的地方是怎样来确定一个整数是偶数还是奇数。通过用模数2对数字执行求模操作， 就是用数字来除以2，并返回余数，从而知道数字是偶数还是奇数。

**更现代的测试版本**

Recent versions of bash include a compound command that acts as an enhanced replacement for test. It uses the following syntax:

目前的 bash 版本包括一个复合命令，作为加强的 test 命令替代物。它使用以下语法：

[[ expression ]]

where, like test, expression is an expression that evaluates to either a true or false result. The [[ ]]command is very similar to test (it supports all of its expressions), but adds an important new string expression:

这里，类似于 test，expression 是一个表达式，其计算结果为真或假。这个[[ ]]命令非常 相似于 test 命令（它支持所有的表达式），但是增加了一个重要的新的字符串表达式：

string1 =~ regex

which returns true if string1 is matched by the extended regular expression regex. This opens up a lot of possibilities for performing such tasks as data validation. In our earlier example of the integer expressions, the script would fail if the constant INT contained anything except an integer. The script needs a way to verify that the constant contains an integer. Using [[ ]] with the =~ string expression operator, we could improve the script this way:

其返回值为真，如果 string1匹配扩展的正则表达式 regex。这就为执行比如数据验证等任务提供了许多可能性。 在我们前面的整数表达式示例中，如果常量 INT 包含除了整数之外的任何数据，脚本就会运行失败。这个脚本 需要一种方法来证明此常量包含一个整数。使用 [[ ]] 和 =~ 字符串表达式操作符，我们能够这样来改进脚本：

#!/bin/bash

# test-integer2: evaluate the value of an integer.

INT=-5

if [[ "$INT" =~ ^-?[0-9]+$ ]]; then

if [ $INT -eq 0 ]; then

echo "INT is zero."

else

if [ $INT -lt 0 ]; then

echo "INT is negative."

else

echo "INT is positive."

fi

if [ $((INT % 2)) -eq 0 ]; then

echo "INT is even."

else

echo "INT is odd."

fi

fi

else

echo "INT is not an integer." >&2

exit 1

fi

By applying the regular expression, we are able to limit the value of INT to only strings that begin with an optional minus sign, followed by one or more numerals. This expression also eliminates the possibility of empty values.

通过应用正则表达式，我们能够限制 INT 的值只是字符串，其开始于一个可选的减号，随后是一个或多个数字。 这个表达式也消除了空值的可能性。

Another added feature of [[ ]] is that the == operator supports pattern matching the same way pathname expansion does. For example:

[[ ]]添加的另一个功能是==操作符支持类型匹配，正如路径名展开所做的那样。例如：

[me@linuxbox ~]$ FILE=foo.bar

[me@linuxbox ~]$ if [[ $FILE == foo.\* ]]; then

> echo "$FILE matches pattern 'foo.\*'"

> fi

foo.bar matches pattern 'foo.\*'

This makes [[ ]] useful for evaluating file and path names.

这就使[[ ]]有助于计算文件和路径名。

**(( )) - 为整数设计**

In addition to the [[ ]] compound command, bash also provides the (( )) compound command, which is useful for operating on integers. It supports a full set of arithmetic evaluations, a subject we will cover fully in Chapter 35.

除了 [[ ]] 复合命令之外，bash 也提供了 (( )) 复合命名，其有利于操作整数。它支持一套 完整的算术计算，我们将在第35章中讨论这个主题。

(( )) is used to perform arithmetic truth tests. An arithmetic truth test results in true if the result of the arithmetic evaluation is non-zero.

(( ))被用来执行算术真测试。如果算术计算的结果是非零值，则一个算术真测试值为真。

[me@linuxbox ~]$ if ((1)); then echo "It is true."; fi

It is true.

[me@linuxbox ~]$ if ((0)); then echo "It is true."; fi

[me@linuxbox ~]$

Using (( )), we can slightly simplify the test-integer2 script like this:

使用(( ))，我们能够略微简化 test-integer2脚本，像这样：

#!/bin/bash

# test-integer2a: evaluate the value of an integer.

INT=-5

if [[ "$INT" =~ ^-?[0-9]+$ ]]; then

if ((INT == 0)); then

echo "INT is zero."

else

if ((INT < 0)); then

echo "INT is negative."

else

echo "INT is positive."

fi

if (( ((INT % 2)) == 0)); then

echo "INT is even."

else

echo "INT is odd."

fi

fi

else

echo "INT is not an integer." >&2

exit 1

fi

Notice that we use less than and greater than signs and that == is used to test for equivalence. This is a more natural looking syntax for working with integers. Notice too, that because the compound command (( )) is part of the shell syntax rather than an ordinary command, and it deals only with integers, it is able to recognize variables by name and does not require expansion to be performed. We’ll discuss (( )) and the related arithmetic expansion further in Chapter 35.

注意我们使用小于和大于符号，以及==用来测试是否相等。这是使用整数较为自然的语法了。也要 注意，因为复合命令 (( )) 是 shell 语法的一部分，而不是一个普通的命令，而且它只处理整数， 所以它能够通过名字识别出变量，而不需要执行展开操作。我们将在第35中进一步讨论 (( )) 命令 和相关的算术展开操作。

**结合表达式**

It’s also possible to combine expressions to create more complex evaluations. Expressions are combined by using logical operators. We saw these in Chapter 18, when we learned about the find command. There are three logical operations for test and [[ ]]. They are AND, OR and NOT. test and [[ ]] use different operators to represent these operations :

也有可能把表达式结合起来创建更复杂的计算。通过使用逻辑操作符来结合表达式。我们 在第18章中已经知道了这些，当我们学习 find 命令的时候。它们是用于 test 和 [[ ]] 三个逻辑操作。 它们是 AND，OR，和 NOT。test 和 [[ ]] 使用不同的操作符来表示这些操作：

|  |  |  |
| --- | --- | --- |
| *Table 28-4: Logical Operators* | | |
| Operation | test | [[ ]] and (( )) |
| AND | -a | && |
| OR | -o | || |
| NOT | ! | ! |

|  |  |  |
| --- | --- | --- |
| *表28-4: 逻辑操作符* | | |
| 操作符 | 测试 | [[ ]] and (( )) |
| AND | -a | && |
| OR | -o | || |
| NOT | ! | ! |

Here’s an example of an AND operation. The following script determines if an integer is within a range of values:

这里有一个 AND 操作的示例。下面的脚本决定了一个整数是否属于某个范围内的值：

#!/bin/bash

# test-integer3: determine if an integer is within a

# specified range of values.

MIN\_VAL=1

MAX\_VAL=100

INT=50

if [[ "$INT" =~ ^-?[0-9]+$ ]]; then

if [[ INT -ge MIN\_VAL && INT -le MAX\_VAL ]]; then

echo "$INT is within $MIN\_VAL to $MAX\_VAL."

else

echo "$INT is out of range."

fi

else

echo "INT is not an integer." >&2

exit 1

fi

We also include parentheses around the expression, for grouping. If these were not included, the negation would only apply to the first expression and not the combination of the two. Coding this with test would be done this way:

我们也可以对表达式使用圆括号，为的是分组。如果不使用括号，那么否定只应用于第一个 表达式，而不是两个组合的表达式。用 test 可以这样来编码：

if [ ! \( $INT -ge $MIN\_VAL -a $INT -le $MAX\_VAL \) ]; then

echo "$INT is outside $MIN\_VAL to $MAX\_VAL."

else

echo "$INT is in range."

fi

Since all expressions and operators used by test are treated as command arguments by the shell (unlike [[ ]] and (( )) ), characters which have special meaning to bash, such as <, >, (, and ), must be quoted or escaped.

因为 test 使用的所有的表达式和操作符都被 shell 看作是命令参数（不像 [[ ]] 和 (( )) ）， 对于 bash 有特殊含义的字符，比如说 <，>，(，和 )，必须引起来或者是转义。

Seeing that test and [[ ]] do roughly the same thing, which is preferable? test is traditional (and part of POSIX), whereas [[ ]] is specific to bash. It’s important to know how to use test, since it is very widely used, but [[ ]] is clearly more useful and is easier to code.

知道了 test 和 [[ ]] 基本上完成相同的事情，哪一个更好呢？test 更传统（是 POSIX 的一部分）， 然而 [[ ]] 特定于 bash。知道怎样使用 test 很重要，因为它被非常广泛地应用，但是显然 [[ ]] 更 有助于，并更易于编码。

Portability Is The Hobgoblin Of Little Minds

可移植性是头脑狭隘人士的心魔

If you talk to “real” Unix people, you quickly discover that many of them don’t like Linux very much. They regard it as impure and unclean. One tenet of Unix followers is that everything should be “portable.” This means that any script you write should be able to run, unchanged, on any Unix-like system.

如果你和“真正的”Unix 用户交谈，你很快就会发现他们大多数人不是非常喜欢 Linux。他们 认为 Linux 肮脏且不干净。Unix 追随者的一个宗旨是，一切都应“可移植的”。这意味着你编写 的任意一个脚本都应当无需修改，就能运行在任何一个类 Unix 的系统中。

Unix people have good reason to believe this. Having seen what proprietary extensions to commands and shells did to the Unix world before POSIX, they are naturally wary of the effect of Linux on their beloved OS.

Unix 用户有充分的理由相信这一点。在 POSIX 之前，Unix 用户已经看到了命令的专有扩展以及 shell 对 Unix 世界的所做所为，他们自然会警惕 Linux 对他们心爱系统的影响。

But portability has a serious downside. It prevents progress. It requires that things are always done using “lowest common denominator” techniques. In the case of shell programming, it means making everything compatible with sh, the original Bourne shell.

但是可移植性有一个严重的缺点。它防碍了进步。它要求做事情要遵循“最低常见标准”。 在 shell 编程这种情况下，它意味着一切要与 sh 兼容，最初的 Bourne shell。

This downside is the excuse that proprietary vendors use to justify their proprietary extensions, only they call them “innovations.” But they are really just lock-in devices for their customers.

这个缺点是一个借口，专有软件供应商用它来证明他们的专利扩展，只有他们称他们为“创新”。 但是他们只是为他们的客户锁定设备。

The GNU tools, such as bash, have no such restrictions. They encourage portability by supporting standards and by being universally available. You can install bash and the other GNU tools on almost any kind of system, even Windows, without cost. So feel free to use all the features of bash. It’s really portable.

GNU 工具，比如说 bash，就没有这些限制。他们通过支持标准和普遍地可用性来鼓励可移植性。你几乎可以 在所有类型的系统中安装 bash 和其它的 GNU 工具，甚至是 Windows，而没有损失。所以就 感觉可以自由的使用 bash 的所有功能。它是真正的可移植。

**控制操作符：分支的另一种方法**

bash provides two control operators that can perform branching. The && (AND) and || (OR)operators work like the logical operators in the [[ ]] compound command. This is the syntax:

bash 支持两种可以执行分支任务的控制操作符。这个 &&（AND）和||（OR）操作符作用如同 复合命令[[ ]]中的逻辑操作符。这是语法：

command1 && command2

and

和

command1 || command2

It is important to understand the behavior of these. With the && operator, command1 is executed and command2 is executed if, and only if, command1 is successful. With the || operator, command1 is executed and command2 is executed if, and only if, command1 is unsuccessful.

理解这些操作很重要。对于 && 操作符，先执行 command1，如果并且只有如果 command1 执行成功后， 才会执行 command2。对于 || 操作符，先执行 command1，如果并且只有如果 command1 执行失败后， 才会执行 command2。

In practical terms, it means that we can do something like this:

在实际中，它意味着我们可以做这样的事情：

[me@linuxbox ~]$ mkdir temp && cd temp

This will create a directory named temp, and if it succeeds, the current working directory will be changed to temp. The second command is attempted only if the mkdir command is successful. Likewise, a command like this:

这会创建一个名为 temp 的目录，并且若它执行成功后，当前目录会更改为 temp。第二个命令会尝试 执行只有当 mkdir 命令执行成功之后。同样地，一个像这样的命令：

[me@linuxbox ~]$ [ -d temp ] || mkdir temp

will test for the existence of the directory temp, and only if the test fails, will the directory be created. This type of construct is very handy for handling errors in scripts, a subject we will discuss more in later chapters. For example, we could do this in a script:

会测试目录 temp 是否存在，并且只有测试失败之后，才会创建这个目录。这种构造类型非常有助于在 脚本中处理错误，这个主题我们将会在随后的章节中讨论更多。例如，我们在脚本中可以这样做：

[ -d temp ] || exit 1

If the script requires the directory temp, and it does not exist, then the script will terminate with an exit status of one.

如果这个脚本要求目录 temp，且目录不存在，然后脚本会终止，并返回退出状态1。

**总结**

We started this chapter with a question. How could we make our sys\_info\_page script detect if the user had permission to read all the home directories? With our knowledge of if, we can solve the problem by adding this code to the report\_home\_space function:

这一章开始于一个问题。我们怎样使 sys\_info\_page 脚本来检测是否用户拥有权限来读取所有的 家目录？根据我们的 if 知识，我们可以解决这个问题，通过把这些代码添加到 report\_home\_space 函数中：

report\_home\_space () {

if [[ $(id -u) -eq 0 ]]; then

cat <<- \_EOF\_

<H2>Home Space Utilization (All Users)</H2>

<PRE>$(du -sh /home/\*)</PRE>

\_EOF\_

else

cat <<- \_EOF\_

<H2>Home Space Utilization ($USER)</H2>

<PRE>$(du -sh $HOME)</PRE>

\_EOF\_

fi

return

}

We evaluate the output of the id command. With the -u option, id outputs the numeric user ID number of the effective user. The superuser is always zero and every other user is a number greater than zero. Knowing this, we can construct two different here documents, one taking advantage of superuser privileges, and the other, restricted to the user’s own home directory.

我们计算 id 命令的输出结果。通过带有 -u 选项的 id 命令，输出有效用户的数字用户 ID 号。 超级用户总是零，其它每个用户是一个大于零的数字。知道了这点，我们能够构建两种不同的 here 文档， 一个利用超级用户权限，另一个限制于用户拥有的家目录。

We are going to take a break from the sys\_info\_page program, but don’t worry. It will be back. In the meantime, we’ll cover some topics that we’ll need when we resume our work.

我们将暂别 sys\_info\_page 程序，但不要着急。它还会回来。同时，当我们继续工作的时候， 将会讨论一些我们需要的话题。

**拓展阅读**

There are several sections of the bash man page that provide further detail on the topics covered in this chapter:

bash 手册页中有几部分对本章中涵盖的主题提供了更详细的内容：

* Lists ( 讨论控制操作符 || 和 && )
* Compound Commands ( 讨论 [[ ]], (( )) 和 if )
* CONDITIONAL EXPRESSIONS （条件表达式）
* SHELL BUILTIN COMMANDS ( 讨论 test )

Further, the Wikipedia has a good article on the concept of pseudocode:

进一步，Wikipedia 中有一篇关于伪代码概念的好文章：

<http://en.wikipedia.org/wiki/Pseudocode>